SFU

More C++

CMPT 125
Mo Chen
SFU Computing Science
11/3/2020

Lecture 24

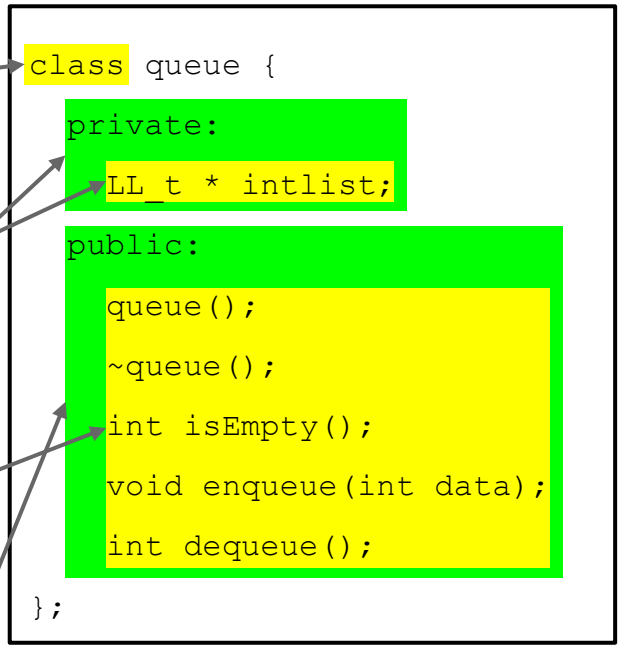
Today:

- C++'inating Your Code
- `new` vs `malloc()`
- `delete` vs `free()`
- Code Re-use

C++ Classes (Review)

C++ uses the keywords `class`, `public:` and `private:` to accomplish *encapsulation* and *information hiding*.

- `class` behaves similar to `struct`, i.e., it declares a *composite data type*
- holds both data members (*properties*) and function members (*methods*)
- mark members with `public:` or `private:` to control their access



(queue.h)

Methods (Review)

Method implementations are denoted by the `class::` prefix

- methods may access all members, `public` or `private`, as if they were local variables

Methods are called by the `object.method()` syntax

```
int queue::isEmpty() {
    return (intlist->head == NULL);
}

void queue::enqueue(int data) {
    LLappend(intlist, data);
}
```

(part of the implementation file `queue.cpp`)

```
queue Q;  // local declaration
Q.enqueue(125);
```

(one option for `driver.cpp`)

```
queue *Q = new queue; // heap decl
Q->enqueue(125);
```

(the other option for `driver.cpp`)

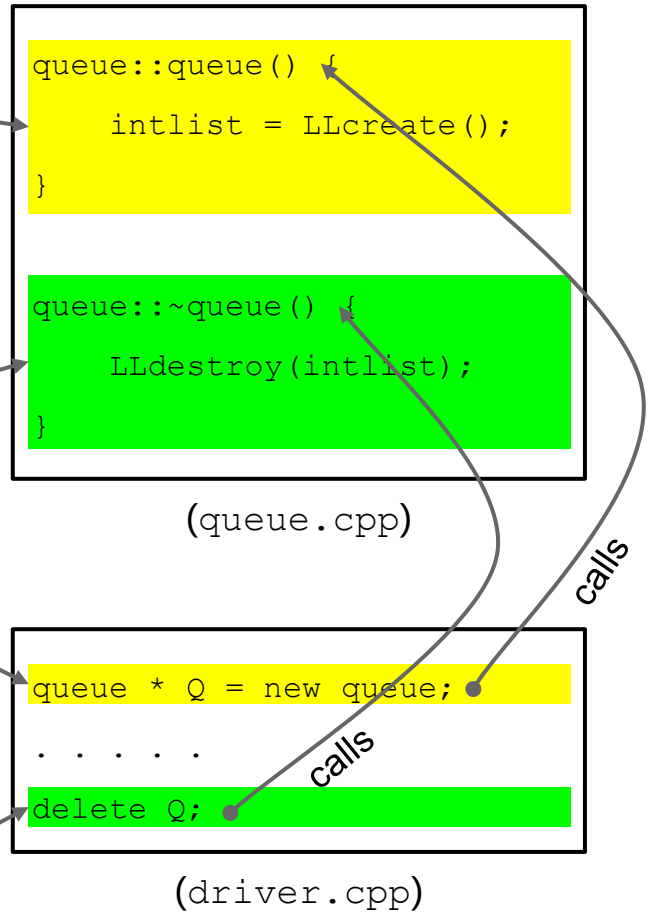
Constructors / Destructors (Review)

A *constructor* is a special method that initializes its data members.

- always called immediately upon instantiation

A *destructor* cleans up any resources held by the object.

- always called when object goes out of scope or is explicitly recycled using `delete`



`malloc()` (Review)

You used `malloc()` for 2 different situations:

Allocate 1 data type.

E.g., `int` or `struct`.

```
int *num = malloc(sizeof(int));
if (num != NULL) *num = 15;

LLnode *n = malloc(sizeof(LLnode));
if (n != NULL) {
    n->data = val;
    n->next = NULL;
}
```

Allocate an array of 1 type.

E.g., a string or an image.

```
char *cpy = malloc(strlen(src)+1);
if (cpy != NULL) strcpy(cpy, src);

uint8_t *pixels = malloc(row*col);
if (pixels != NULL) {
    . . .
    . . .
}
```

In almost all cases, you initialized immediately after allocating the space.

`new` vs `malloc()`

The `new` operator isn't just for instantiating objects: it does all that `malloc()` does.

- but for objects, it also runs the constructor method

```
int *num = malloc(sizeof(int));
if (num != NULL) *num = 15;
```

→

```
int *num = new int;
if (num != NULL) *num = 15;
```

```
LLnode *n = malloc(sizeof(LLnode));
if (n != NULL) {
    n->data = val;
    n->next = NULL;
}
```

→

```
LLnode *n = new LLnode;
if (n != NULL) {
    n->data = val;
    n->next = NULL;
}
```

```
char *cpy = malloc(strlen(src)+1);
```

→

```
char *cpy = new char[strlen(src)+1];
```

`new` vs `malloc()`

The `new` operator isn't just for instantiating objects: it does all that `malloc()` does.

- but for objects, it also runs the constructor method

```
int *num = malloc(sizeof(int));
if (num != NULL) *num = 15;
```

→

```
int *num = new int(15);
```

```
LLnode *n = malloc(sizeof(LLnode));
if (n != NULL) {
    n->data = val;
    n->next = NULL;
}
```

→

```
LLnode *n = new LLnode;
if (n != NULL) {
    n->data = val;
    n->next = NULL;
}
```

```
char *cpy = malloc(strlen(src)+1);
```

→

```
char *cpy = new char[strlen(src)+1];
```

`new` vs `malloc()`

The `new` operator isn't just for instantiating objects: it does all that `malloc()` does.

- but for objects, it also runs the constructor method

```
int *num = malloc(sizeof(int));
if (num != NULL) *num = 15;
```

→

```
int *num = new int(15);
```

```
LLnode *n = malloc(sizeof(LLnode));
if (n != NULL) {
    n->data = val;
    n->next = NULL;
}
```

→

```
LLnode *n = new LLnode(val, NULL);
```

```
char *cpy = malloc(strlen(src)+1);
```

→

```
char *cpy = new char[strlen(src)+1];
```

Function Overloading

Multiple versions of functions may be useful

- Especially for constructors

Example: Constructor for the queue class

- Current version: create an empty queue

```
public:
    queue();
```

- Sometimes, creating a queue with one element is convenient

```
public:
    queue(int data);
```

Function Overloading

Both (and in general, all) versions of functions can be implemented simultaneously by overloading the function

- Use the same function name
- Use different input parameters

```
public:
    queue();
    queue(int data);
```

- The version of the function that gets executed depends on how the function is called

Function Overloading

queue.h

```
class queue {
  private:
    LL_t * intlist;
  public:
    queue();
    queue(int data);
    ~queue();
    int isEmpty();
    void enqueue(int data);
    int dequeue();
};
```

queue.cpp

```
...
queue::queue() {
    intlist = Llcreate();
}

queue::queue(int data) {
    intlist = Llcreate();
    LLappend(intlist, data);
}
...
```

Another Example

Employee class

- Properties: name, ID, job title, salary
- Methods:
  - employee, ~employee, promote, demote, fire, give_raise
  - set_name, set_ID, set_job_title, set_salary
  - get_name, get_ID, get_job_title, get_salary

- Considerations for constructor
  - Create “empty” employee, then use set_ functions to populate properties:

```
employee * joe_gupta = new employee();
joe_gupta->set_name("Sumeet Gupta");
joe_gupta->set_salary(314159);
```

  - Create employee with name and salary

```
employee * joe_gupta = new employee("Sumeet Gupta", 314159);
```

Another Example

```
class employee {
    private:
        legal_name
        id
        job_title
        salary

    public:
        employee();
        employee(char * legal_name, int salary);
        ~employee();
        promote();
        demote();
        fire();
        give_raise(int new_salary);

        set_name(char * legal_name);
        set_ID(int id);
        set_job_title(char * job_title);
        set_salary(int salary);

        char * get_name();
        int get_ID();
        char * get_job_title();
        int get_salary();
}
```

`delete` vs `free()`

Both return allocated space to the heap, where:

- `free()` is the inverse of `malloc()`
- `delete` is the inverse of `new`
- `delete []` is the inverse of `new []`
- `delete` and `delete []` run the destructor before recycling

```
LLnode *n = new LLnode(val, NULL);
. . .
. . .
delete n;
```

```
char *cpy = new char[strlen(src)+1];
. . .
. . .
delete [] cpy;
```

Code Re-Use

If a piece of code can be employed for multiple purposes, then you *factor* the code

- Principle: Write it once, and then re-use it.

These are interfaces, but taken to the next level:

- libraries (E.g., `stdio.h`, `stdlib.h`, STL)
- design patterns (E.g., object oriented design)
- frameworks (E.g., Bootstrap, Cocoa, .Net, QX)

Rule of Thumb: Avoid cut & paste

- Updates and debugging won't affect other versions.

A Queue of Integers

The Story So Far:

- We just developed a Queue ADT which . . .
- depended on a Linked List ADT which . . .
- depended on a Node . . .

but it only works for integers.

What if we wanted a queue of . . .

- doubles?
- strings?
- ordered pairs?

Generic Programming

Express the algorithms so that they work on *any* type, to be specified as a parameter.

C++ uses the `template` construct to do this.

```
class queue {
  private:
    LL_t * intlist;
  public:
    queue();
    ~queue();
    int isEmpty();
    void enqueue(int data);
    int dequeue();
};
```

```
template <class T>
class queue {
  private:
    LL_t * intlist;
  public:
    queue();
    ~queue();
    int isEmpty();
    void enqueue(int data);
    int dequeue();
};
```

Generic Programming

Express the algorithms so that they work on *any* type, to be specified as a parameter.

C++ uses the `template` construct to do this.

```
class queue {
  private:
    LL_t * intlist;
  public:
    queue();
    ~queue();
    int isEmpty();
    void enqueue(int data);
    int dequeue();
};
```

```
template <class T>
class queue {
  private:
    Linked List of Ts;
  public:
    queue();
    ~queue();
    int isEmpty();
    void enqueue(T data);
    T dequeue();
};
```